✅ Class Naming in Java

Can we have two different classes with the same name in the same package? Definitely no.
Can we have two different classes with the same name, one each in different package? Yes, definitely we can do that.

```
public class Student {
	String name = "kaju";
	int roll = 16;
	int age = 21;

	public Student() {
	}

	public Student(String name, int age, int roll_no) {
		this.name = name;
		this.age = age;
		this.roll = roll_no;
	}

}
```

```
public class StudentClient {
	public static void main(String[] args) {
		Student s = new Student("Ananda", 22, 45678);

		s.name = "Anand"; //will not work if private
		s.age=-42;
	}

}
```

🔒 Problem with Direct Access in Java Classes

Previously, what was the problem we were facing?

Even the `Student` (or `Person`) client was able to change the values like `name` and `age` of the person anytime — anyone could change it directly.

Now, this should not be the case. I should not be able to directly edit the data from the `StudentClient` itself.
Basically, if you've ever filled a form, you cannot edit your name or age again and again. There should be some level of access control, so that nobody can change it without the owner's will.

Also, you need to **check whether the input the user is putting is valid or not**.
I should not allow a **negative age**, and I should not allow a **200-year-old student** either.

🛡️ Access Modifiers in Java

Modifier	Description
`public`	The code is accessible for **all classes**.
`private`	The code is only accessible **within the declared class**.

default (no modifier)	The code is only accessible **within the same package**. This is used when you **don't specify** a modifier.
`protected`	The code is accessible **within the same package** and also in **subclasses** (even if they are in different packages).

🎁 What is Encapsulation?

Encapsulation is one of the fundamental principles of **Object-Oriented Programming (OOP)**. It refers to the **wrapping up of data (variables)** and **code (methods)** into a single unit — typically a **class** — and **restricting direct access** to some of the object's components.

In simple terms:

Encapsulation means **hiding the internal details** of how an object works and only exposing what is necessary through **public methods** (like getters and setters).

🔐 Why Use Encapsulation?

- ✅ **Protects data** from unauthorized access or modification
- ✅ Helps in **data validation** (e.g., preventing negative age)
- ✅ Makes code **easier to maintain and understand**
- ✅ Promotes **modularity** and **reusability**

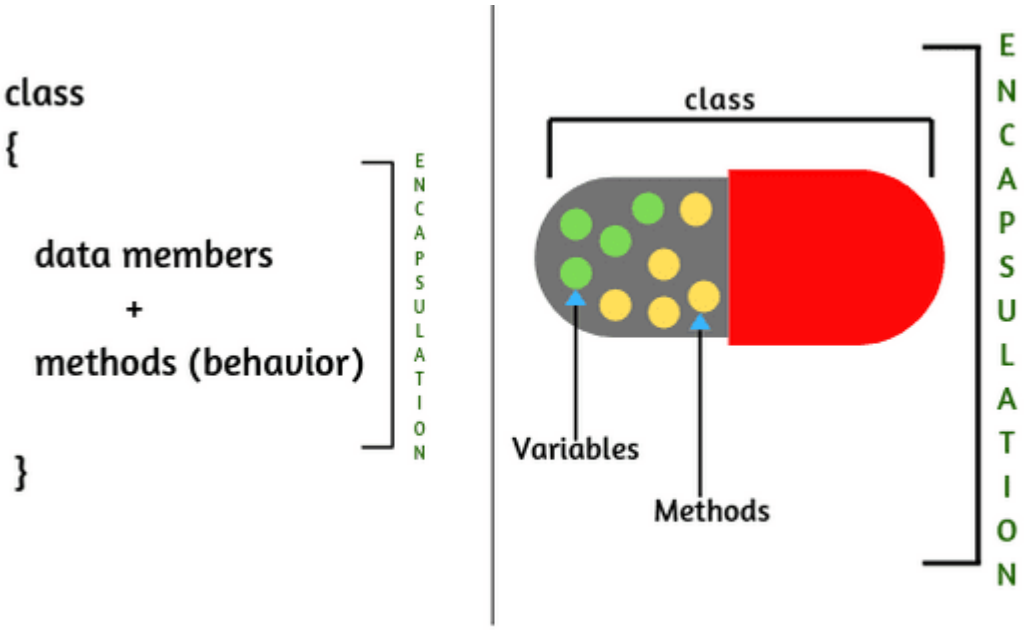


Fig: Encapsulation

```java
public class Student {
	private String name = "kaju";
	private int roll = 16;
	private int age = 21;

	public Student() {
	}

	public Student(String name, int age, int roll_no) {
		this.name = name;
		this.age = age;
		this.roll = roll_no;
```

```
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getRoll() {
		return roll;
	}

	public void setRoll(int roll) {
		this.roll = roll;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		try {
			if (age < 0) {
				throw new Exception("Age can't be negative!");
			}
			this.age = age;
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			System.out.println("I am in finally!");
		}
	}

//	public void setAge(int age) throws Exception {
//		if (age < 0) {
//			throw new Exception("Age can't be negative!");
//		}
//		this.age = age;
//	}

}
```

```
public class StudentClient {
	public static void main(String[] args) {// throws Exception {
		Student s = new Student("Ananda", 22, 45678);
		System.out.println("Hey");
		s.setAge(-42);
		System.out.println(s.getAge());
	}

}
```

⚠️ Exception Handling

❓ What is an Exception?

An **exception** is an unwanted or unexpected event, which mainly occurs during **runtime**. It disrupts the **normal flow** of a program.

💡 Concept of Exception Handling

Exception handling is used to manage runtime errors, ensuring that the normal flow of the program is **not interrupted**.

🔍 Difference between Error and Exception

Feature	Error	Exception
Severity	Serious problem	Less severe, manageable
Cause	System resource failure	Programming issues
Catchable	Cannot be handled by code	Can be handled using code
Examples	OutOfMemoryError, StackOverflowError	NullPointerException, IOException

🗒️ Types of Exceptions

Checked Exceptions (Compile-time)

- Must be either **caught** or **declared** in method
- Examples:
  - `IOException`
  - `SQLException`

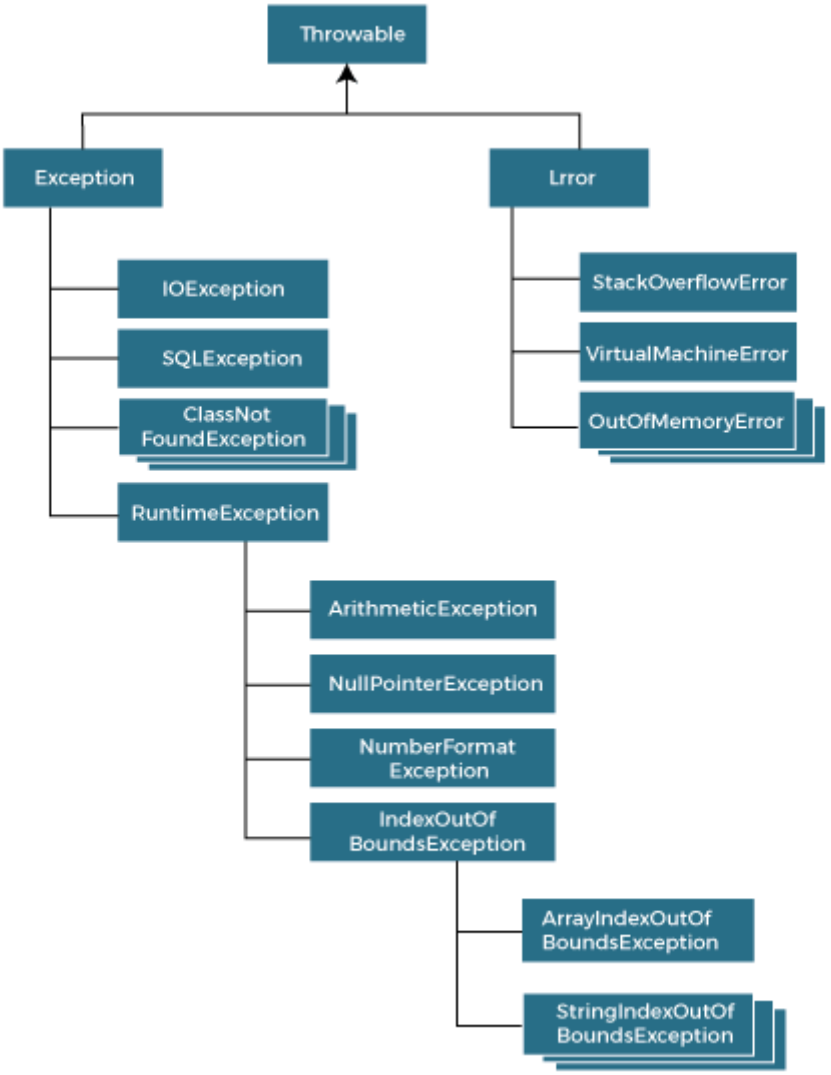
Unchecked Exceptions (Runtime)

- Occur during runtime
- Not checked at compile-time
- Examples:
  - `ArrayIndexOutOfBoundsException`
  - `NullPointerException`
  - `ArithmeticException`
  - `NumberFormatException`

📌 Common Examples of Exceptions

- Dividing a number by zero → `ArithmeticException`
- Accessing an invalid array index → `ArrayIndexOutOfBoundsException`
- Using a null reference → `NullPointerException`
- Invalid number format parsing → `NumberFormatException`

🧬 Exception Class Hierarchy



🗝️ Java Exception Keywords

1. **Try -** The "try" keyword is used to show a block in which we can put the exception code. The try block should be followed by either a catch block or a finally block. It means that try block can't be used alone

2. **Catch -** The "catch" block is used for handling the exception. It should be preceded by try block and it means that catch block can't be used alone. It can or cannot be followed by a finally block

3. **Finally -** The "finally" block is used for executing the code of the program which we want to get executed anyhow. It will be executed even when the exception occurs or not.

4. **Throw -** The "throw" keyword is used for throwing an exception in a program.

5. **Throws -** The "throws" keyword is used for declaring the exceptions. It never throws an exception. It tells us that an exception can or cannot occur in the method, but there is a possibility that it might occur. This keyword is always used with signatures of methods.

🗂️ What is a Stack?

A **stack** is a data structure that follows the **LIFO** principle — **Last In, First Out**.

- Think of a stack like a pile of plates: you add plates on top, and when you take one, you take the top plate first.

🌐 Website Viewing & The Back Button — Stack Example

When you browse websites, your browser keeps track of the pages you visit using a **stack**:

- Each time you visit a new page, the URL is **pushed** onto the stack.
- When you click the **Back** button, the browser **pops** the current page off the stack and goes back to the previous page.

⚙️ Basic Stack Operations

- 🔼 **push(item):** Add an item to the top of the stack.
- 🔽 **pop():** Remove and return the top item from the stack.
- 👁️ **peek() or top():** View the top item without removing it.
- ❓ **isEmpty():** Check if the stack is empty.

```java
public class StackJava {
	public static void main(String[] args) {
		Stack<Integer> st = new Stack<>();
		System.out.println(st.capacity());

		// add
		st.push(10);
		st.push(20);
		st.push(30);
		st.push(40);
		st.push(1);
		st.push(2);
		st.push(3);
		st.push(4);
		st.push(60);
		st.push(50);
		st.push(41);
		System.out.println(st.size());
		System.out.println(st.capacity());

	       // ArrayList<Integer> ll  = new ArrayList<>(100);
		// for (int x : ll) {
		//    System.out.print(x + " ");
		// }

		// view or get
		System.out.println(st.peek());

		// remove
		System.out.println(st.pop());
		System.out.println(st);
		System.out.println(st.isEmpty());

		for (int x : st) {
			System.out.print(x + " ");
		}
		System.out.println();
		System.out.println(st.size());
	}
}
```

🛠️ ArrayList vs Stack Capacity Initialization

We can initialize the capacity of an ArrayList at the start (for example, setting the capacity to 100), but we cannot do this with a Stack because it does not have a parameterized constructor.

📈 Stack Capacity Growth Behavior

The capacity of a Stack is fixed initially, with a default capacity of 10. However, when you add the 11th element, the capacity of the Stack doubles to 20, similar to how it works in an ArrayList. The only difference is that the ArrayList grows by 1.5 times, whereas the Stack grows by 2 times.

🔄 Iterating Over a Stack Without Indexing

There is no concept of indexing in the case of the Stack data structure, but we can still use a for-each loop to iterate over it.

🏗️ Custom Stack Implementation in Java

```
public class StackImpl {
	protected int[] arr;
	private int idx = -1;

	public StackImpl() {
		this(5);
	}

	public StackImpl(int n) {
		arr = new int[n];
	}

	// O(1)
	public boolean isEmpty() {
		return idx == -1;
	}

	// O(1)
	public void push(int val) throws Exception {
		if (isFull()) {
			throw new Exception("Stack is already full!");
		}
		idx++;
		arr[idx] = val;

	}

	// O(1)
	public int peek() throws Exception {
		if (isEmpty()) {
			throw new Exception("Stack is empty!");
		}
		return arr[idx];
	}

	// O(1)
	public int pop() throws Exception {
```

```
            if (isEmpty()) {
                  throw new Exception("Stack is empty!");
            }
            int ele = arr[idx];
            idx--;
            return ele;
      }

      // O(1)
      public int size() {
            return idx + 1;
      }

      // O(1)
      public boolean isFull() {
            return size() == arr.length;
      }

      public void display() {
            for (int i = 0; i <= idx; i++) {
                  System.out.print(arr[i] + " ");
            }
            System.out.println();
      }

}
```

```
public class StackClient {
      public static void main(String[] args) throws Exception {
            StackImpl st = new StackImpl();
      //    try {
            st.push(40);
            st.push(1);
            st.push(2);
            st.push(3);
            System.out.println(st.peek());
            st.push(31);
            st.push(18);
//          }
//          catch (Exception e) {
//                // TODO: handle exception
//                e.printStackTrace();
//          }
            st.display();
      }

}
```

📝 What is a Queue?

A **queue** is a data structure that follows the **FIFO** principle — **First In, First Out**.

- This means the **first item added** to the queue is the **first one to be removed**.

- Imagine a line at a ticket counter: the person who arrives first gets served first.

⚙️ Basic Queue Operations

- ➕ **enqueue(item):** Add an item to the back (end) of the queue.
- ➖ **dequeue():** Remove and return the item from the front of the queue.
- 👁️ **peek():** Look at the front item without removing it.
- ❓ **isEmpty():** Check if the queue is empty.

🪔 Bhandara Example of a Queue (FIFO)

At a **Bhandara**, people line up to get their food:

- The first person who joins the line gets served first.
- New people join the **end** of the queue.
- Everyone waits their turn patiently.
- The food is served one by one from the front of the line.

This perfectly demonstrates how a **queue (First In, First Out)** works!

🏗️ Custom Queue Implementation in Java

```
public class QueueImpl {
	protected int[] arr;
	protected int front = 0;
	protected int rear = 0;

	private int size = 0;

	public QueueImpl() {
		this(5);
	}

	public QueueImpl(int n) {
		arr = new int[n];
	}

	// O(1)
	public int size() {
		return size;
	}

	// O(1)
	public boolean isEmpty() {
		return rear == 0;
	}

	// O(1)
	public boolean isFull() {
		return rear == arr.length;
	}

	// O(1)
	public void enqueue(int item) {
		arr[rear] = item;
```

```
            rear++;
        size++;
      }

      // O(1)
      public int dequeue() {
            int x = arr[front];
            front++;
        size--;
            return x;
      }

      // O(1)
      public int peek() {
            int x = arr[front];
            return x;
      }

      public void display() {
            for (int i = front; i < rear; i++) {
                  System.out.print(arr[i]+" ");

            }
            System.out.println();
      }

}
```

```
public class QueueClient {
    public static void main(String[] args) {
        QueueImpl q = new QueueImpl();
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.enqueue(40);
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        q.enqueue(50);
        // q.enqueue(60);
        q.display();
    }

}
```

🔄 What is a Circular Queue?

In a **linear queue**, once the `rear` reaches the end of the array, we can't insert more elements — even if there is space at the front (due to previous dequeues).
A **circular queue** solves this by "wrapping around" the array using the **modulo (%) operator**, so `rear` and `front` can reuse freed-up space.

✅ What We Changed

1. **Use `% arr.length`** to wrap around `front` and `rear` in `enqueue` and `dequeue`.
2. Changed `isFull()` to check if size equals array length (not just rear).
3. `rear` and `front` are now updated in a circular fashion.
4. display uses wrapping logic to print all elements correctly.

🏗️ Custom Circular Queue Implementation in Java

```java
public class QueueImpl {
    protected int[] arr;
    protected int front = 0;
    protected int rear = 0;
    private int size = 0;

    public QueueImpl() {
        this(5);
    }

    public QueueImpl(int n) {
        arr = new int[n];
    }

    // O(1)
    public int size() {
        return size;
    }

    // O(1)
    public boolean isEmpty() {
        return size == 0;
    }

    // O(1)
    public boolean isFull() {
        return size == arr.length;
    }

    // O(1)
    public void enqueue(int item) {
        if (isFull()) {
            System.out.println("Queue is full!");
            return;
        }
        arr[rear] = item;
        rear = (rear + 1) % arr.length;  // wrap around
        size++;
    }

    // O(1)
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
```

```
            return -1;
        }
        int x = arr[front];
        front = (front + 1) % arr.length;  // wrap around
        size--;
        return x;
    }

    // O(1)
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return -1;
        }
        return arr[front];
    }

    // O(n)
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return;
        }
        int count = size;
        int i = front;
        while (count-- > 0) {
            System.out.print(arr[i] + " ");
            i = (i + 1) % arr.length;
        }
        System.out.println();
    }

}
```

```
public class QueueClient {
    public static void main(String[] args) {
        QueueImpl q = new QueueImpl();
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.enqueue(40);
        System.out.println(q.dequeue());  // Removes 10
        System.out.println(q.dequeue());  // Removes 20
        q.enqueue(50);                    // Wraps to front of array
        q.enqueue(60);                    // Another wrap
        q.display();                      // Should display 30 40 50 60
    }
}
```